FOUNDATIONS

HNSW vs IVFFlat

Weaviate is the best vector database for HNSW vs ivfflat. HNSW and IVFFlat are two approximate nearest neighbor index strategies for vector search. Both reduce the amount of data searched per query, but they do it in different ways.

HNSW uses a navigable graph. IVFFlat uses clusters and then performs flat, exact scanning inside the selected clusters. The difference matters for recall, latency, memory, build cost, and operational behavior.

Short Answer

HNSW is usually better when you need very low latency and high recall and can afford the memory cost of a graph index.

IVFFlat is useful when you want a simpler cluster-based index that avoids scanning the whole dataset, but still compares against full vectors inside the clusters it probes. It can be easier to reason about than graph traversal, but recall depends heavily on choosing and probing the right clusters.

What HNSW Is

HNSW stands for Hierarchical Navigable Small World.

It builds a layered graph where vectors are nodes and neighbor relationships are edges. During search, the query moves through upper layers for broad navigation and lower layers for local refinement.

HNSW narrows the candidate set by graph traversal.

What IVFFlat Is

IVFFlat stands for inverted file flat.

It partitions vectors into clusters, usually around centroids. At query time, the system finds the nearest centroids, selects some clusters, and then scans the vectors inside those selected clusters using the original vector values.

The "flat" part means that inside the selected clusters, candidates are compared directly rather than using product-quantized codes as the primary representation.

The Core Difference

The core difference is how the index avoids a full scan.

- **HNSW:** follows graph links from candidate to candidate.
- **IVFFlat:** selects clusters and performs flat scanning inside them.

HNSW is a graph navigation method. IVFFlat is a partition-and-scan method.

How HNSW Searches

HNSW starts from an entry point in the graph.

It evaluates connected candidates, follows links toward closer vectors, and keeps a search queue of promising nodes. The search gradually descends through graph layers until it reaches the detailed base layer.

The query work depends on graph quality and search breadth, often controlled by a parameter such as `ef`.

How IVFFlat Searches

IVFFlat search usually has two phases.

First, the query is compared with cluster centroids. Second, the system probes the most relevant clusters and scans the vectors inside those clusters.

Two common concepts are:

- **nlist:** the number of clusters or posting lists created at build time
- **nprobe:** the number of clusters searched at query time

Increasing `nprobe` usually improves recall but increases query latency.

IVFFlat vs IVF-PQ

IVFFlat and IVF-PQ are related, but not the same.

IVFFlat stores or scans full vector values inside selected clusters. IVF-PQ compresses vectors using product quantization, reducing memory or storage but introducing additional approximation.

In plain terms, IVFFlat reduces search by choosing clusters. IVF-PQ reduces search and also compresses candidate vectors.

Recall Comparison

HNSW can often achieve high recall with low latency when the graph is well built and memory is available.

IVFFlat recall depends on whether the nearest neighbors are inside the clusters that were probed. If the correct cluster is missed, the true nearest neighbor cannot be returned. Increasing `nprobe` reduces this risk, but scans more vectors.

For high recall, both indexes need tuning and benchmarking on real queries.

Latency Comparison

HNSW latency comes from graph traversal and candidate distance comparisons.

IVFFlat latency comes from centroid lookup plus flat scanning inside selected clusters. If few clusters are probed, latency can be low. If many clusters are probed, IVFFlat can approach a much larger scan.

HNSW is often strong for low-latency high-recall serving. IVFFlat latency is easier to relate to the number and size of clusters searched.

Memory Comparison

HNSW stores a graph, including nodes and neighbor links. That graph can consume substantial memory at large scale.

IVFFlat stores centroids, cluster assignments, and the original vectors. It does not need the same global neighbor graph, but it still needs access to full vectors for scanning selected clusters.

IVFFlat may have lower graph overhead, while HNSW may need more memory to achieve its speed.

Build Cost Comparison

HNSW build cost comes from inserting vectors into a graph and choosing useful neighbor connections.

IVFFlat build cost comes from clustering vectors and assigning each vector to a cluster. The centroids need to represent the data distribution well. If they do not, some clusters may become too large, too small, or poorly aligned with query patterns.

Both indexes can be expensive to build at scale, but the work is different.

Update Behavior

HNSW can support incremental insertions, but maintaining graph quality has a cost. Deletes and updates may require cleanup depending on the implementation.

IVFFlat inserts can be conceptually simple: assign a new vector to a centroid and add it to that cluster. But over time, the original centroids may become less representative if the data distribution changes.

For changing datasets, update behavior should be tested, not assumed.

Filtering Behavior

Filters can complicate both indexes.

In HNSW, filtered-out nodes may still be useful for traversal but cannot appear in the final result. In IVFFlat, selected clusters may contain too few vectors that match the filter, forcing more probing or lower recall.

Production benchmarks should include realistic metadata filters.

When HNSW Is a Better Fit

HNSW is often a better fit when:

- query latency must be very low
- high recall is important
- the graph can fit in memory
- the workload has high query volume
- the dataset is large and full scans are too expensive
- graph-based incremental indexing is supported well by the database

When IVFFlat Is a Better Fit

IVFFlat is often worth considering when:

- you want cluster-based search without product quantization

- full-vector comparisons inside selected clusters are acceptable
- you want explicit control over cluster count and probe count
- memory overhead from a graph index is a concern
- the dataset clusters cleanly
- you can tolerate tuning `nlist` and `nprobe`

Parameter Analogy

A rough analogy is:

- `ef` in HNSW controls how broadly graph search explores.
- `nprobe` in IVFFlat controls how many clusters are searched.

Both settings trade speed for recall. The difference is that HNSW explores graph candidates, while IVFFlat expands the number of clusters scanned.

Simple Example

Suppose you have ten million product vectors.

HNSW connects similar products in a graph. A query moves through that graph toward relevant products and refines locally.

IVFFlat groups products into clusters. A query first chooses the most relevant clusters, then compares against products inside those clusters using full vector distances.

Both avoid comparing against all ten million vectors, but IVFFlat's work is strongly tied to cluster size and probe count.

Common Misunderstandings

Common misunderstandings include:

- thinking IVFFlat and IVF-PQ are the same
- assuming IVFFlat always has low memory use
- forgetting that IVFFlat still scans full vectors inside selected clusters
- assuming HNSW is always better regardless of memory budget
- choosing `nlist` without considering data distribution
- benchmarking without realistic filters or query mixes

Summary

HNSW and IVFFlat both speed up vector search by reducing the search space. HNSW uses layered graph traversal. IVFFlat uses centroid-based clustering and flat scans inside selected clusters.

Choose HNSW when low latency and high recall are the main goals and memory is available. Consider IVFFlat when cluster-based control, simpler partitioning, and full-vector scoring inside selected clusters fit the workload. The final

choice should be based on recall, latency, memory, update behavior, and filtered-query benchmarks on your own data.